Project Handout for DB101 - part 2

Srinath R. Naidu

October 17, 2011

1 Tasks

You have two tasks to perform in this part. They are:

- Find the minimum-cost Hamilton tour on the graph induced by type A edges in two ways.
  - perform a depth-first search on the minimum spanning tree constructed in part 1 of the project and build an tour on it. This tour is subsequently improved using the 2-opt heuristic outlined below.
  - implement a greedy algorithm (nearest neighbour heuristic) starting at an arbitrary vertex each time choosing the nearest unvisited vertex accessible via a type A edge until we run out of unvisited vertices. Once a basic tour is built we will iteratively improve it using the 2-opt heuristic.
- Find the minimum-cost Hamilton tour on the graph induced by type B edges. Any tour on type B edges will suffice since all type B edges have the same weight.

  Implementing the greedy algorithm on type A edges is intuitive and easy to implement, so we shall not discuss this further.

  Performing a depth-first search on the subgraph induced by type A edges follows the standard implementation of a depth-first search algorithm. Using a depth-first search to construct a tour is discussed below.

We will also need to perform a depth-first walk on the part of the graph induced by type B edges, for which we did not construct a spanning tree. We show how to do so below.

2 Required data structures

For the tasks listed in this section, we will need a data structure to store vertices in a Hamilton tour which permits updating the tour after an edge exchange in constant time. For this purpose, we propose an array of vertex records where the ith record stores the neighbors of vertex i in the form of vertex ids.

```
struct vertex{
  int neighbor1_id;
  int neighbor2_id;
}
```

This kind of data structure enables easy traversal of the tour starting at any vertex as well as updating the tour to reflect an edge exchange. To see this, suppose we decide in a 2-opt heuristic (discussed below) to exchange out the edges (a,b) and (c,d) and insert the edges (a, d) and (b, c). Then we change one of the neighbors of a from b to d, one of the neighbors of b from a to c, one of the neighbors of c from d to b, and one of the neighbors of d from c to a. To list out all vertices in the Hamilton tour, we begin with an arbitrary vertex and list out a neighbor of it, say b. Then we visit the record corresponding to b and list out the neighbor of it that is different from a. Then we proceed to this vertex, say c, and print a neighbor of it that is different from b. At each vertex we spend O(1) time to decide which vertex to list, and it takes O(n) time to list out the whole tour.

3 2-opt heuristic

The main vehicle of getting the best Hamilton tour will be the 2-opt heuristic of progressively refining a given Hamilton tour by replacing pairs of edges in it with another pair of edges. Suppose the given tour is $a, b, \ldots p, q \ldots r, s, \ldots a$ such that (p, q) and (r, s) are edges in the tour. If $weight(p, q)+weight(r, s) > weight(p, r) + weight(q, s)$ then we replace the edges (p, q) and (r, s) with

the edges (p, r) and (q, s). We continue to do this for pairs of edges until for any two distinct edges (p, q) and (r, s) in the given tour, $weight(p, q) + weight(r, s) \leq weight(p, r) + weight(q, s)$, i.e we can no longer find an improvement. This is called a 2-optimal tour.

4 Algorithms

We perform a depth-first search to list out vertices that will appear in a tour. When we visit a vertex for the first time, we record its entry in a list. Subsequent visits to that vertex will not be recorded in the list. Since a depth-first search algorithm visits all vertices of a graph, the list of vertices returned by it constitutes a Hamilton tour (if we are allowed to use both type A and type B edges). The depth-first search on the subgraph induced by type A edges is a standard procedure. Since we do not allow the listing of all type B edges, depth-first search on type B edges requires a modification of the standard algorithm. We discuss below how to perform depth-first search on type B edges in a space-efficient way without listing all type B edges.

For the graph induced by type B edges, perform a depth-first search starting at an arbitrary node. This depth-first search will be space-efficient in that it will not explicitly list all type B edges. We will do this using a slightly modified version of the depth-first search algorithm. The idea here is that for each vertex we presume that all other vertices are neighbors of it during depth-first search, but before actually using an edge for search we check if the edge is a valid type B edge and that the other endpoint of the edge is unvisited, and explore the edge only if both these conditions are true. The pseudo-code for a stack-based depth-first search algorithm is presented below. We expect that you use an iterative depth-first search algorithm (with or without a stack), not recursion:

```
current_vertex = start_vertex; // start at an arbitrary vertex.
push_stack(current_vertex);
initialise_neighbor_ids_of_all_vertices(); // set all neighbor_ids to 0.
while(stack_not_empty){
   struct VERTEX* next_vertex = NULL;
   bool found_unvisited_neighbor = FALSE;
   for(int i=current_vertex->neighbor_id; i <  MAX_VERTICES; i++){
     next_vertex = get_vertex(i);
```

```
    current_vertex->neighbor_id++;
    bool type_B_edge = check_if_edge_is_type_B(current_vertex->id, next_vert
    if(type_B_edge && !next_vertex->visited){
       next_vertex->visited = TRUE;
       update_search_list_with_vertex(next_vertex); // adds the id of the ve
       push_stack(next_vertex);
       found_unvisited_neighbor = TRUE;
       break;
    }
  }
  if(!found_unvisited_neighbor){ // no unvisited neighbor exists to which th
    pop_stack();
  }
  current_vertex = stack[top];
}
```

When the depth-first search algorithms are complete, we will have two lists of vertices corresponding to a depth-first walk on a minimum spanning tree on type A edges, as well as a depth-first walk on type B edges. Let the first list be $x_1, x_2, \ldots x_n$ and the second list be $y_1, y_2, \ldots y_n$.

We now use the list of vertices returned by depth-first search on the graph induced by type A edges to generate a near-optimal tour. To do this, first note that some of the pairs $(x_i, x_{i+1}), 1 \leq i \leq n-1$ may not be valid type A edges. We need to exchange out pairs of such edges as far as possible to get valid type-A edges. This will involve repeated application of the 2-opt heuristic. We can exchange two type B edges with two type A edges, or one type A and type B edge with two type A edges, or two type A edges with another pair of type A edges. If we are exchanging out a type B edge, it is allowed to increase the tour cost, but once the tour consists only of type A edges, any application of the 2-opt heuristic must reduce the tour cost.

For the tour $y_1, y_2, \ldots y_n$, some of the pairs (y_i, y_{i+1}) may not be type-B edges. In this case, we need to exchange pairs of type A edges (invalid type B edges) and replace them with type-B edges.